ECE 6775
High-Level Digital Design Automation
Fall 2023

Analysis of Algorithms

Cornell University

CSL

Announcements

- HW 1 will be released tomorrow

Agenda

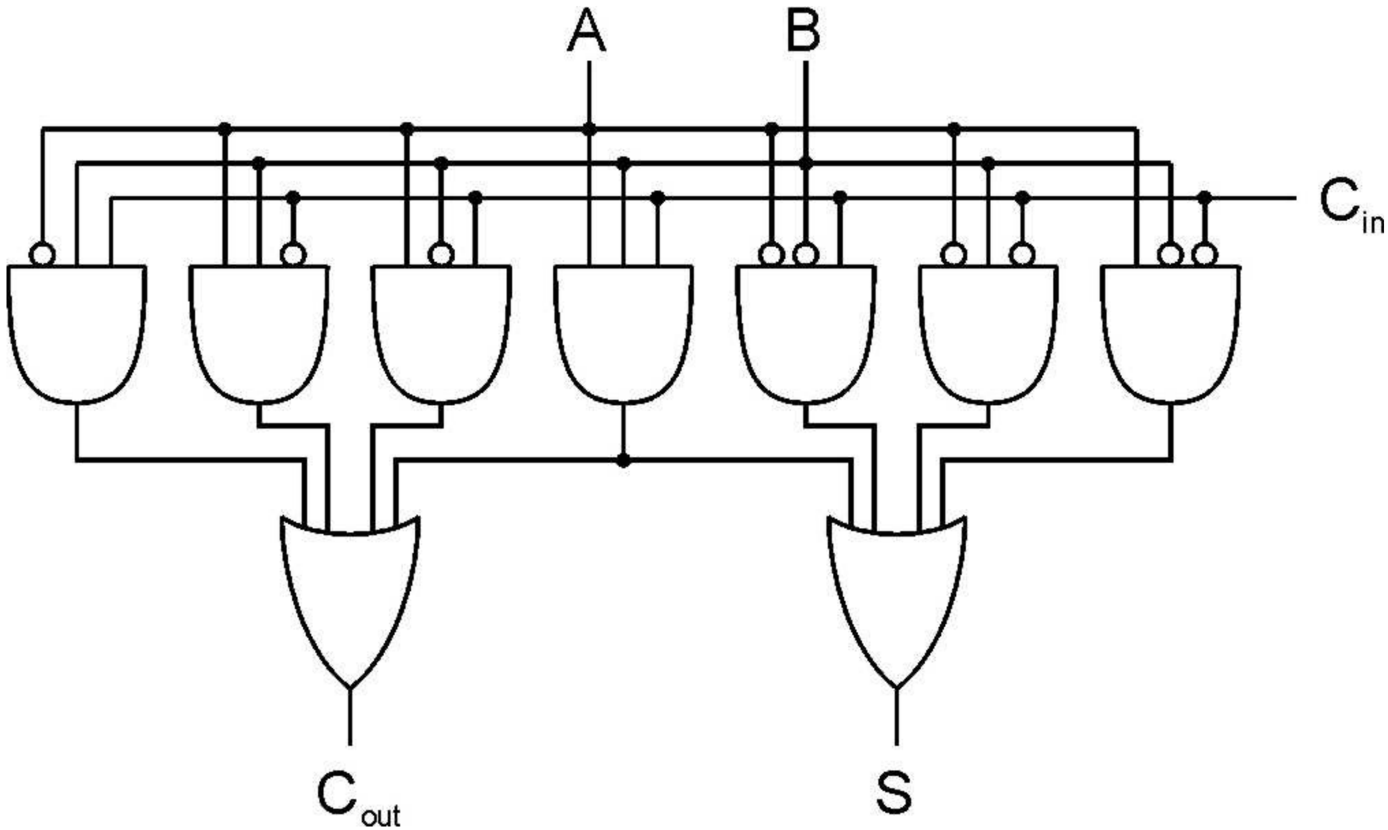
- Basics of algorithm analysis
  - Complexity analysis and asymptotic notations
  - Taxonomy of algorithms
- Basics of graph algorithms
  - EDA application: Static timing analysis

Review: LUT Mapping

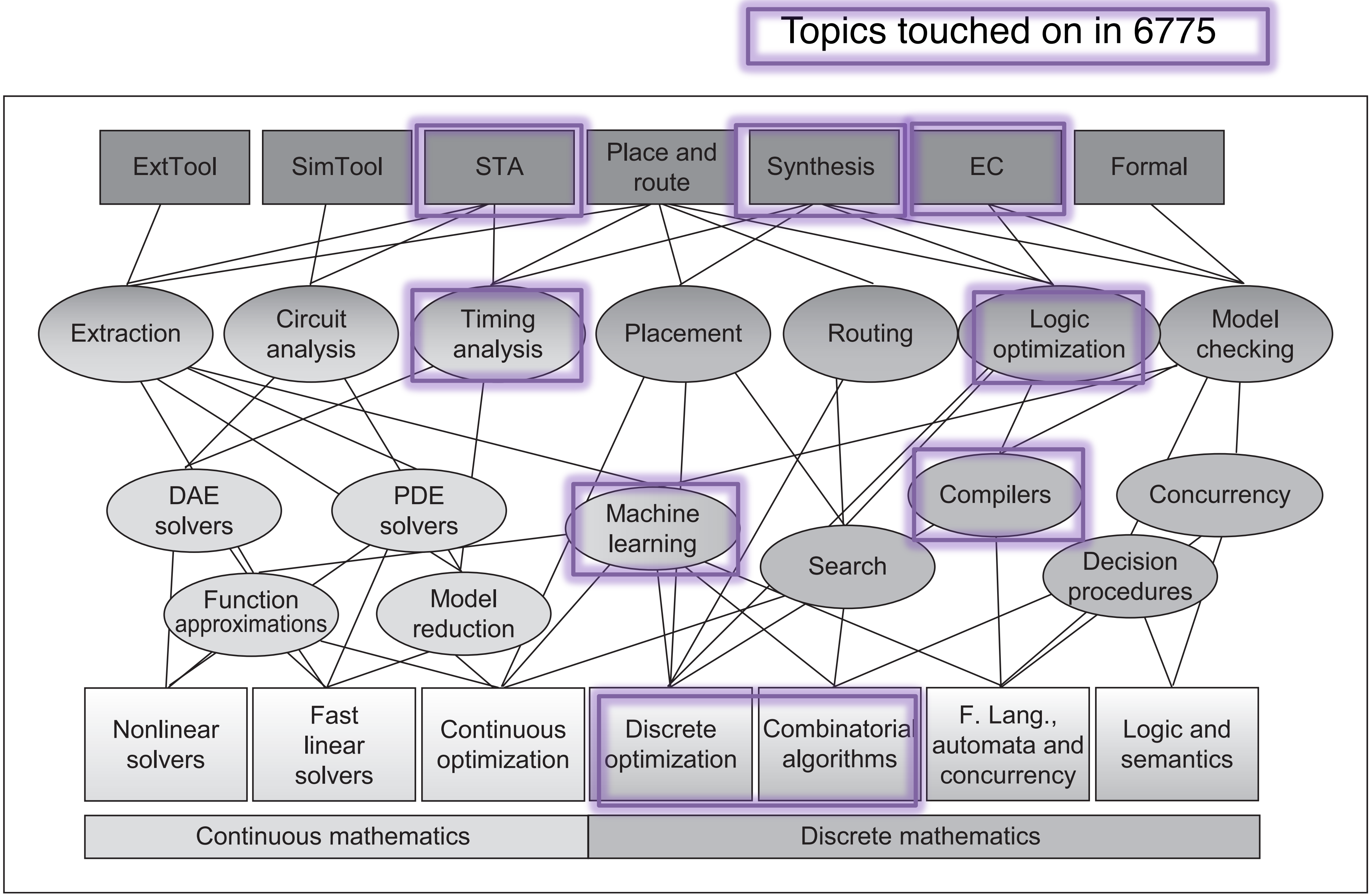
(1) How many 3-input LUTs are needed to implement the following full adder?

(2) How about using 4-input LUTs?

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Recap: Algorithms Drive Automation



Key Algorithms in EDA

[source: Andreas Kuehlmann, Synopsys Inc.]

Analysis of Algorithms

- Need a systematic way to compare two algorithms
  - Execution time is typically the most common criterion used
  - Space (memory) usage is also important in most cases
  - But difficult to compare in practice since these algorithms may be implemented on different machines, use different languages, etc.
  - Plus, execution time is usually input-dependent

- **big-O** notation is widely used for asymptotic analysis
  - Complexity is represented with respect to some natural & abstract measure of the problem size *N*

Big-O Notation

- Express execution time as a function of input size *n*
  - Running time F(n) is of order G(n), written as F(n) is **O**(G(n)) when $\exists n_0, \forall n \geq n_0, F(n) \leq K \cdot G(n)$ for some constant *K*
  - F will not grow larger than G by more than a constant factor
  - G is often called an "**upper bound**" for F
- Interested in the worst-case input & the growth rate for large input size

Big-O Notation (cont.)

- How to determine the order of a function?
  - Ignore lower order terms
  - Ignore multiplicative constants
  - Examples:

    $3n^2 + 6n + 2.7$ is $O(n^2)$

    $n^{1.1} + 10000000000n$ is $O(n^{1.1})$, $n^{1.1}$ is also $O(n^2)$

    $n! > C^n > n^C > \log n > \log \log n > C$

    $\Rightarrow n! > n^{10}$; $n \log n > n$; $n > \log n$

- What do asymptotic notations mean in practice?
  - If algorithm A is $O(n^2)$ and algorithm B is $O(n \log n)$,

    we usually say algorithm B is **more scalable**.

More Asymptotic Notions

- **big-Omega** notation: F(n) is $\Omega(G(n))$
  - $\exists n_0, \forall n \geq n_0, F(n) \geq K \cdot g(n)$ for some constant K
    G is called a "**lower bound**" for F
- **big-Theta** notation: F(n) is $\Theta(G(n))$
  - If G is both an upper and lower bound for F, it describes the growth of a function more accurately than big-O or big-Omega
  - Examples:
    $4n^2 + 1024 = \Theta(n^2)$
    $n^3 + 4n \neq \Theta(n^2)$

Exponential Growth

- Consider a 1 GHz processor (1 ns per clock cycle) running 2^n operations (assuming each op requires one cycle)

n	2^n	1ns (/op) x 2^n
10	10^3	1 us
20	10^6	1 ms
30	10^9	1 s
40	10^{12}	16.7 mins
50	10^{15}	11.6 years
60	10^{18}	31.7 years
70	10^{21}	31710 years

NP-Complete

- The class **NP-complete** (NPC) is the set of decision problems which we “believe” there is no polynomial time algorithms (hardest problem in NP)

- **NP-hard** is another class of problems, which are at least as hard as the problems in NPC (also containing NPC)

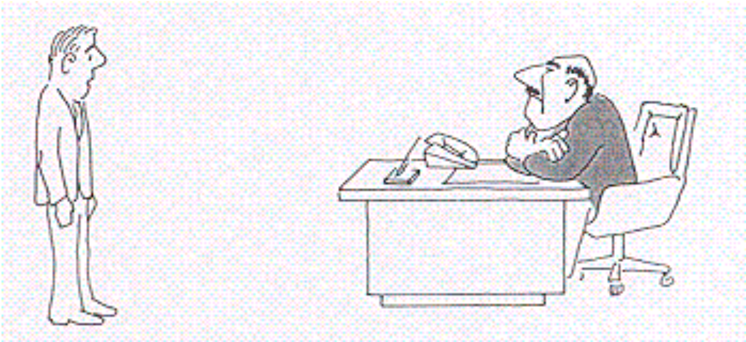
- If we know a problem is in NPC or NP-hard, there is (very) little hope to solve it exactly in an efficient way

Reduction

- Showing a problem P is at least as hard as (or not easier than) another problem Q
  - Formal steps:
    - Given an instance q of problem Q,
      there is a polynomial-time transformation to an instance p of P,
      q is a "yes" instance if and only if p is a "yes" instance
  - Informally, if P can be solved efficiently, we can solve Q efficiently **(Q is reduced to P)**
    - P is polynomial time solvable → Q is polynomial time solvable
    - Q is not polynomial time solvable → P is not polynomial time solvable
- Example
  - Problem P: Sort *n* numbers
  - Problem Q: Given *n* numbers, find the median

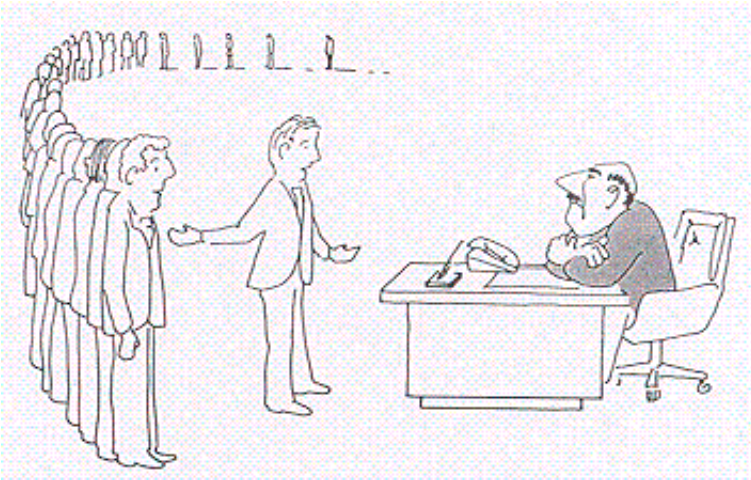
How to Identify an NP-Complete Problem

- I can't find an efficient algorithm, I guess I'm just too dumb.



- I can't find an efficient algorithm, because no such algorithm is possible.

- I can't find an efficient algorithm, but neither can all these famous people.



[source: "Computers and Intractibility" by Garey and Johnson]

Types of Algorithms

- There are many ways to categorize different types of algorithms
  - Polynomial vs. Exponential, in terms of computational effort
  - Optimal (or Exact) vs. Heuristic, in solution quality
  - Deterministic vs. Stochastic, in decision making
  - Constructive vs. Iterative, in structure

  …

Problem Intractability

- Most of the nontrivial EDA problems are intractable (NP-complete or NP-hard)
  - Best-known algorithm complexities that grow exponentially with n, e.g., O(n!), $O(n^n)$, and $O(2^n)$.
  - No known algorithms can ensure, in a time-efficient manner, globally optimal solution

- **Heuristic** algorithms are used to find near-optimal solutions
  - Be content with a “reasonably good” solution

Many Algorithm Design Techniques

- There can be many different algorithms to solve the same problem
  - Exhaustive search
  - Divide and conquer
  - Greedy
  - Dynamic programming
  - Network flow
  - ILP
  - Simulated annealing
  - Evolutionary algorithms

  …

Broader Classification of Algorithms

- Combinatorial algorithms
  - **Graph algorithms**

  …
- Computational mathematics
  - **Optimization algorithms**
  - Numerical algorithms

  …
- Computational science
  - Bioinformatics
  - Linguistics
  - Statistics

  …
- Digital logic
  - Boolean minimization

  …
- Information theory & signal processing

…

- Machine learning and statistical classification

Topics touched on in 6775

Many more

[source: en.wikipedia.org/wiki/List_of_algorithms]

Graph Definition

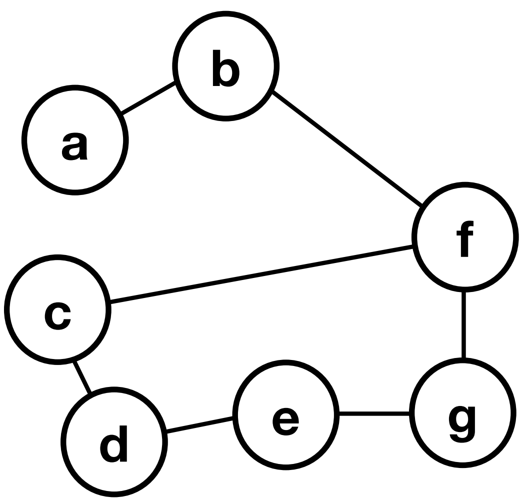
- Graph: a set of objects and their connections
  - Ubiquitous: any binary relation can be represented as a graph

- Formal definition:
  - $G = (V, E)$, $V = \{v_1, v_2, ..., v_n\}$, $E = \{e_1, e_2, ..., e_m\}$
    - V : set of **vertices** (nodes), E : set of **edges** (arcs)
  - **Undirected graph**: an edge {u, v} also implies {v, u}
  - **Directed graph**: each edge (u, v) has a direction

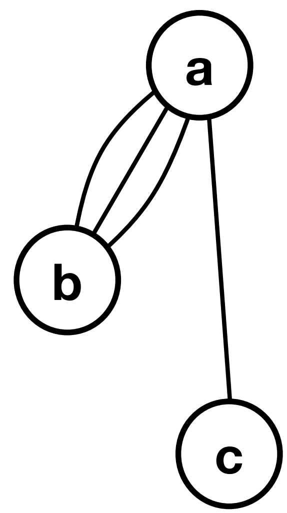
Simple Graph

- Loops, multi edges, and simple graphs
  - An edge of the form (v, v) is said to be a **self-loop**
  - A graph permitted to have multiple edges (or parallel edges) between two vertices is called a **multigraph**
  - A graph is said to be **simple** if it contains no self-loops or multiedges

Simple graph



Multigraph



Graph Connectivity

- Paths
  - A **path** is a sequence of edges connecting two vertices
  - A **simple path** never goes through any vertex more than once

- Connectivity
  - A graph is **connected** if there is there is a path between any two vertices
  - Any subgraph that is connected can be referred to as a **connected component**
  - A directed graph is **strongly connected** if there is always a directed path between vertices

Trees and DAGs

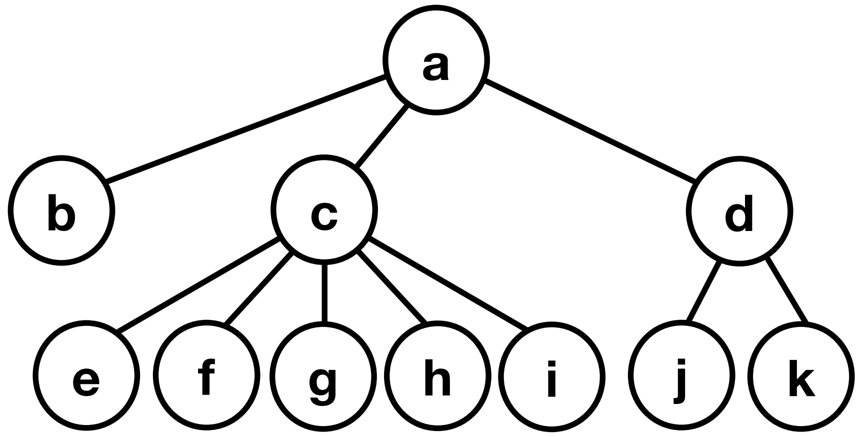
- A **cycle** is a path starting and ending at the same vertex. A cycle in which no vertex is repeated other than the starting vertex is said to be a **simple cycle**

- An undirected graph with no cycles is a **tree** if it is connected, or a **forest** otherwise
  - A **directed tree** is a directed graph which would be a tree if the directions on the edges were ignored

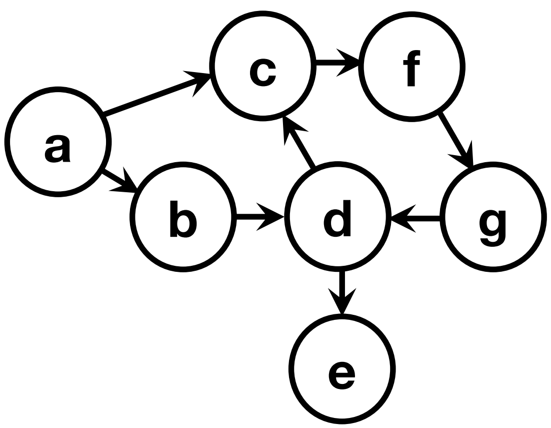
- A directed graph with no directed cycles is said to be a **directed acyclic graph** (**DAG**)

Examples

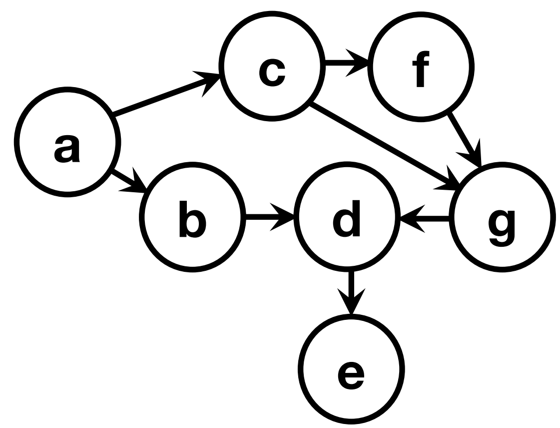
Tree



Directed graphs with cycles



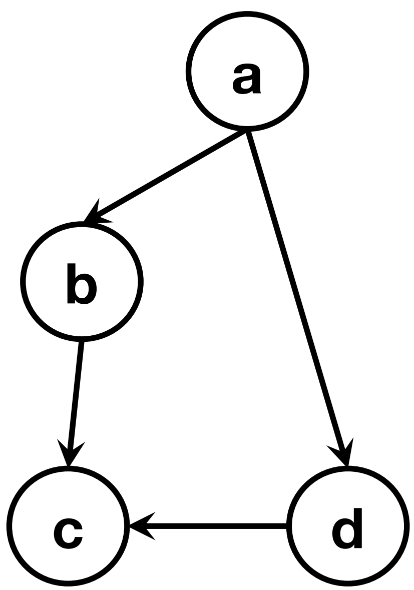
Directed acyclic graph (DAG)



Graph Traversal

- Purpose: visit all the vertices in a particular order, check/update their properties along the way

- Commonly used algorithms
  - Depth-first search (DFS)
  - Breadth-first search (BFS)

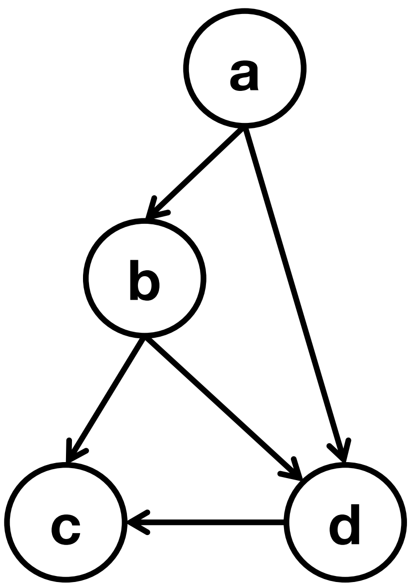


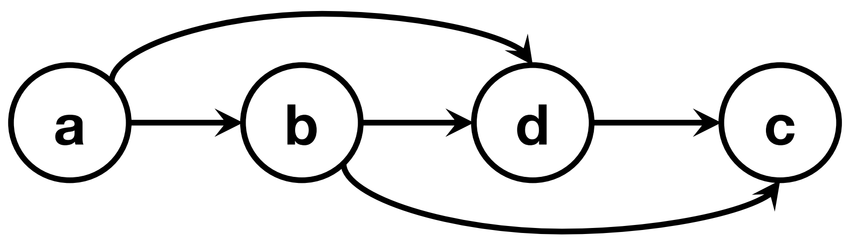
DFS order (from node a): ??

BFS order: ??

Topological Sort

- A **topological order of a directed graph** is an ordering of nodes where all edges go from an earlier vertex (left) to a later vertex (right)
  - Feasible if and only if the subject graph is a DAG





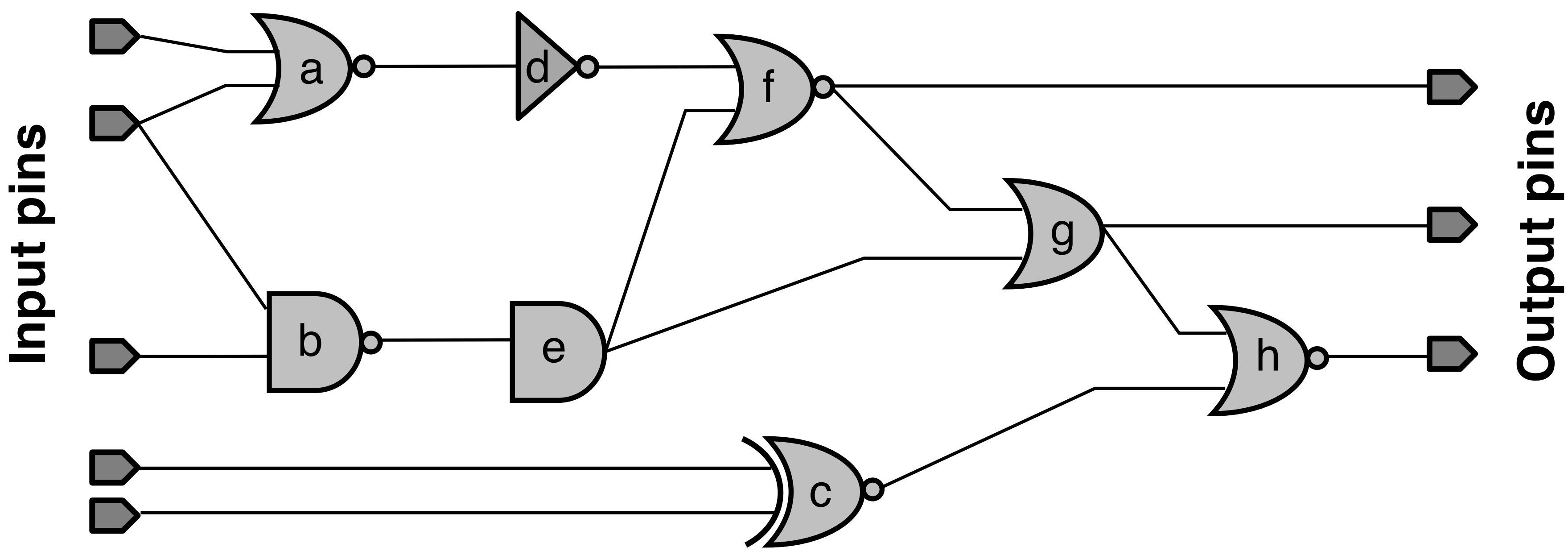
Application in EDA: Static Timing Analysis

- In circuit graphs, **static timing analysis** (STA) refers to the problem of finding the delays from the input pins of the circuit (esp. nodes) to each gate
  - In sequential circuits, flip-flop (FF) input acts as output pin, FF output acts as input pin
  - Max delay of the output pins determines clock period
  - **Critical path** is a path with max delay among all paths

- Two important terms
  - **Required time**: The time that the data signal needs to arrive at certain endpoint on a path to ensure the timing is met
  - **Arrival time**: The time that the data signal actually arrives at certain endpoint on a path

STA: An Example

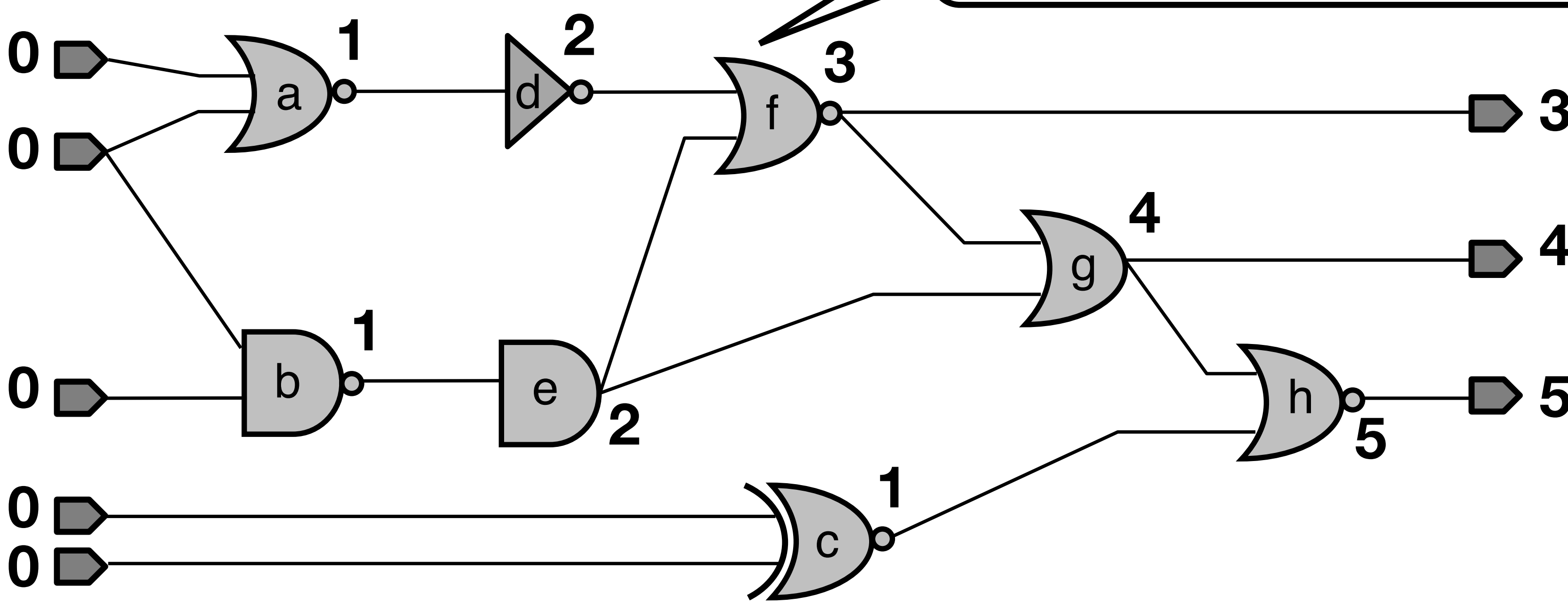
- **pred**(n): predecessors of node n
  - e.g., **pred**(f) = {d, e}
- **succ**(n): successors of node n
  - e.g., **succ**(e) = {f, g}



STA: Arrival Times

- Assumptions
  - All inputs arrive at time 0
  - All gate delays = 1ns (d = 1); all wire delays = 0
- Questions: **Arrival time** (AT) of each gate output? Minimum clock period?

$AT_f = \max_{k \in pred(f)}\{AT_k\} + d_f$

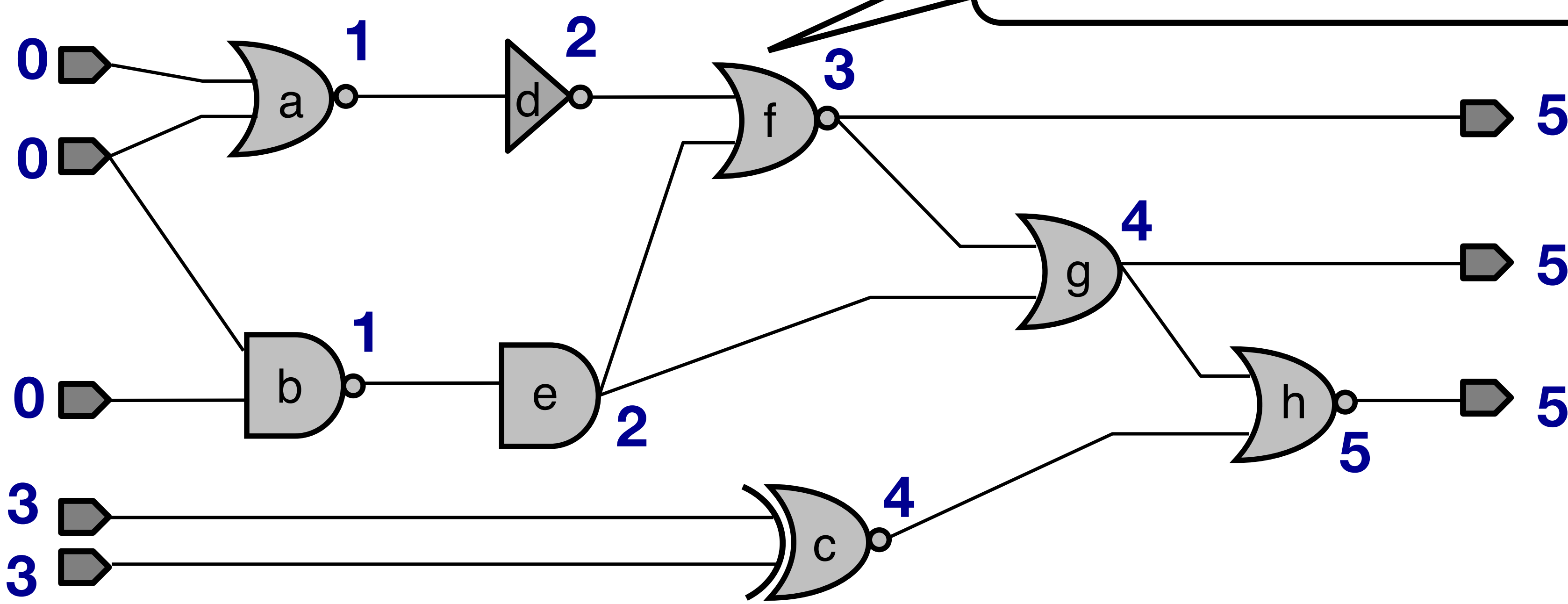


Gates are visited in a topological order

STA: Required Times

- Assumptions
  - All inputs arrive at time 0
  - All gate delays = 1ns (d = 1); all wire delays = 0
  - Clock period = 5ns (200MHz frequency)
- Question: **Required time** (RT) of each gate output in order to meet the clock period?

$RT_f = \min_{k \in succ(f)} \{RT_k - d_k\}$



Gates are visited in a reverse topological order

STA: Slacks

- In addition to the arrival time and required time of each node, we are interested in knowing the **slack** (= RT - AT) of each node / edge
  - Negative slacks indicate unsatisfied timing constraints
  - Positive slacks often present opportunities for additional (area/power) optimization
  - Node on the **critical path** have zero slacks

Next Lecture

- Binary decision diagrams (BDDs)

Acknowledgements

- These slides contain/adapt materials from / developed by
  - Prof. David Pan (UT Austin)
  - “VLSI Physical Design: From Graph Partitioning to Timing Closure” authored by Prof. Andrew B. Kahng, Prof. Jens Lienig, Prof. Igor L. Markov, Dr. Jin Hu